ORM 框架动态实体设计需求文档

1. 背景与问题陈述

本项目旨在开发一个创新的 ORM（对象关系映射）框架，专注于解决企业级应用，特别是 **ERP** 系统 在 多租户、高定制化和频繁扩展 场景下所面临的实体模型管理挑战。

传统的 ERP 系统通常通过预先设计并生成强类型实体类来承载业务数据。然而，在云原生和高度可配置的环境中，这种方法带来了严重的灵活性和维护性问题。

1.1 早期动态实体方案及其局限

为了应对传统强类型实体在扩展性上的不足，项目早期采用了运行时动态实体的设计。这种方案通过 DynamicObjectType 定义实体结构，并使用 DynamicObject 实例配合 object[] 来存储数据。

早期方案的优势 **(选择原因):**

- 高度灵活性：无需修改或重新编译现有强类型实体，即可在运行时动态添加或修改字段。
- 简化扩展管理：避免了传统强类型实体在多重、平行或依赖扩展场景下复杂的继承关系管理。

早期方案存在的问题 **(当前痛点):**

尽管解决了扩展性问题，但这种设计引入了显著的性能和开发体验问题：

1. 内存消耗大：原始类型（如 long, int, bool）在存入 object[] 时需要进行装箱 **(Boxing)** 操作。这导致每个原始类型值都变成一个对象，额外增加了大量的内存开销，远超强类型实体。
2. **CPU** 消耗大：
   - 数据存取时频繁发生装箱/拆箱 **(Boxing/Unboxing)** 操作，增加了 CPU 负担。
   - 通过字符串名称访问属性（例如 entity.setValue("orderId", 1)）在内部通常需要进行字典查找 **(Dictionary Lookup)**，这比强类型实体的直接字段访问效率低得多。
3. 开发体验不佳：
   - 属性访问依赖于字符串硬编码，容易在开发过程中写错，且缺乏编译时检查。
   - 当属性名需要修改时，重构困难，IDE 无法提供有效的支持，容易引入运行时错误。

2. 传统强类型实体在扩展性上遇到的实际问题

在 ERP 这种高度可定制的领域，如果坚持使用传统的强类型实体，会遇到以下实际且难以解决的问题：

1. 复杂的继承关系和版本管理：
    - 当原厂发布 SalesOrder 实体并基于此实现业务逻辑后，如果需要为 SalesOrder 开发一个“行业扩展”增加字段，通常需要创建 SalesOrderExtension1 继承自 SalesOrder。
    - 如果又有一个独立的“平行扩展”也需要扩展 SalesOrder，则可能产生 SalesOrderExtension2。
    - 问题在于，当一个租户同时安装了多个扩展时，如何将这些扩展字段整合到一个统一的强类型实体中？例如，SalesOrderExtension1 和 SalesOrderExtension2 如何共同作用于同一个 SalesOrder 实例？这会导致复杂的多重继承（如果语言支持）或组合模式，难以管理和维护。
    - 从数据库加载的数据类型（SalesOrder vs SalesOrderExtension1 vs SalesOrderExtension2）以及如何安全地传递给原厂程序，都是巨大的挑战。
2. 实体数量爆炸：
    - 一个典型的 ERP 系统可能包含 2 万个核心实体。
    - 在公有云环境下，一个 Docker 容器可能为 100 个租户提供服务，且 Java 进程是共享的。
    - 如果每个租户的每个扩展都生成独立的强类型实体类，那么在一个共享的 Java 进程中，将存在天文数字般的实体类。这将对 JVM 的类加载器和 **Metaspace**（元空间）造成巨大压力，导致性能下降甚至崩溃。

3. 新方案的实际思路

为了解决上述所有问题，新方案将采用一种混合策略，结合了运行时动态性、编译时类型安全和内存优化。

3.1 核心设计理念

- 运行时动态创建实体类作为**“数据承载体”**：在最终用户的执行环境，根据实际需要动态生成底层的实体类，这些类将仅作为数据的物理存储结构，类似于 .NET 的 ValueTuple。
- 设计时生成接口实现强类型访问：为了解决开发体验问题，允许插件开发人员在扩展实体时，通过工具生成对应的 Java 接口，业务代码可以基于这些接口进行类型安全的访问。

3.2 具体优化策略

1. 内存与 **CPU** 效率优化：
    - 避免装箱/拆箱：动态生成的实体类将直接使用 Java 的原始类型（int, long, boolean 等）或 Object 类型来存储字段值，从而消除装箱/拆箱的性能开销和内存浪费。
    - 直接字段访问：底层动态生成的类将拥有直接的字段，属性访问将是直接的字段访问，而非字典查找，显著提升 CPU 效率。

2. 开发体验提升：
    - 接口驱动的强类型访问：插件开发人员在扩展实体（如 SalesOrder）时，会通过工具差量化记录新增字段，并生成对应的 Java 接口。开发人员可以将这些接口复制到自己的业务代码中，实现对扩展字段的强类型、编译时安全的访问。
    - 支持重构：通过接口访问，IDE 可以提供更好的重构支持。
3. 解决“类爆炸”问题 **(关键)**：
    - **“数据承载体”**原则：动态生成的实体类将严格遵循“数据承载体”的原则，它们不包含业务逻辑，仅用于存储数据。
    - 基于字段类型组合的唯一性生成：
        - Java 的原始类型有 8 种 (byte, short, int, long, float, double, char, boolean)。
        - 所有引用类型（如 String, UUID, 自定义对象等）在底层统一视为 Object 类型。
        - 因此，底层数据承载体类的泛型参数类型集合为：{byte, short, int, long, float, double, char, boolean, Object} (共 9 种)。
        - ORM 框架将仅为实际业务中出现的、独特的字段类型组合生成一个唯一的底层类。例如，如果 SalesOrder 需要 (int, int, String) 字段组合，而 PurchaseOrder 也需要 (int, int, String)，则只会生成一个 Tuple_I_I_O 类型的内部类。
    - 大实体分解为小元组：对于字段数量非常多的实体（例如 150-200 个字段的 SalesOrder），不会生成一个包含所有字段的单一类。相反，这些字段将按照类型排序后，分组为多个小的**“元组”**实例（例如，每 7 个字段一组，类似于 .NET ValueTuple 的设计 ）。一个逻辑上的 SalesOrder 实体将由多个这样的底层元组实例组合而成。
      ValueTuple 的设计思路是为了提供一种轻量、高效的方式来组合多个不同类型的值，而无需定义专门的类。它通过提供一系列预定义的泛型结构（例如 ValueTuple<T1, T2>, ValueTuple<T1, T2, T3, T4, T5, T6, T7>）来实现。当需要组合的字段数量超过预设的最大值（例如 7 个）时，ValueTuple 巧妙地利用其最后一个泛型参数 TRest 来引用另一个 ValueTuple 实例，从而形成一个链式结构，实现对任意数量字段的组合。这种设计使得每个元组实例都保持较小的固定大小，避免了为每个字段组合都生成一个新类的开销，同时又能高效地存储和访问数据，并且由于是值类型，还能减少垃圾回收的压力。

      参见：
      https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/ValueTuple.cs
    - **Metaspace** 压力缓解：通过上述策略，即使逻辑实体数量庞大（2 万个），实际在 JVM Metaspace 中加载的动态生成的类数量将大大减少（例如，可能只有几千种独特的字段组合类 ），从而有效控制 Metaspace 的大小，确保系统在多租户共享进程环境下的稳定性和性能。

3.3 ORM 框架的职责边界

除了核心的实体创建和访问，该 ORM 框架还将承担以下职责：

- 复杂查询支持：能够理解动态实体结构并构建高效的数据库查询。
- 缓存管理：支持基于动态实体结构的缓存机制。
- 事务管理：提供完整的事务支持。
- 界面数据承载：作为 MVC 架构中 Model 层的底层数据承载，支持 UI 层的双向数据绑定和操作。